Adaptive indexing for relational keys

Goetz Graefe, Harumi Kuno

HP Laboratories
HPL-2010-23

Keyword(s):
databases, indexes, storage systems, B-trees, adaptive merging, database cracking

Abstract:

Adaptive indexing schemes such as database cracking and adaptive merging have been investigated to-date only in the context of range queries. These are typical for non-key columns in relational databases. For complete self-managing indexing, adaptive indexing must also apply to key columns. The present paper proposes a design and offers a first performance evaluation in the context of keys. Adaptive merging for keys also enables further improvements in B-tree indexes. First, partitions can be matched to levels in the memory hierarchy such as a CPU cache and an in-memory buffer pool. Second, adaptive merging in merged B-trees enables automatic master-detail clustering.

External Posting Date: February 6, 2010 [Fulltext]   Approved for External Publication
Internal Posting Date: February 6, 2010 [Fulltext]

To be published and presented at SMDB 2010, Long Beach, CA March 1, 2010

Adaptive indexing for relational keys[1]

Goetz Graefe[1], Harumi Kuno[2]

HP Labs
Palo Alto, CA USA
[1]goetz.graefe@hp.com
[2]harumi.kuno@hp.com

***Abstract*— Adaptive indexing schemes such as database cracking and adaptive merging have been investigated to-date only in the context of range queries. These are typical for non-key columns in relational databases. For complete self-managing indexing, adaptive indexing must also apply to key columns. The present paper proposes a design and offers a first performance evaluation in the context of keys.**

Adaptive merging for keys also enables further improvements in B-tree indexes. First, partitions can be matched to levels in the memory hierarchy such as a CPU cache and an in-memory buffer pool. Second, adaptive merging in merged B-trees enables automatic master-detail clustering.

I. Introduction

Adaptive indexing should apply to key columns just as well as to non-key columns. However, neither of the two techniques for adaptive indexing, database cracking and adaptive merging, has been designed for this context.

The purpose of the present research is to explore adaptive indexing in the context of key columns. Specifically, the paper proposes a design for adaptive merging, explores additional applications of adaptive merging for indexes on keys, and presents a preliminary performance evaluation of database cracking and adaptive merging in this context.

Database cracking [11-13] pioneered adaptive indexing. New indexes are created and optimized as a side effect of query execution, with fairly low cost and automatic focus on the key ranges searched in actual queries. For example, if most or all queries search for information relevant to the most recent months, the index is never optimized for prior months.

Adaptive merging [6] is a second adaptive indexing technique. As in database cracking, an index covers all rows in a table, index creation and index optimization are side effects of query execution, and optimization effort focuses on key ranges searched in actual queries. Unlike database cracking, adaptive merging uses standard data structures (B-trees) and algorithms (run generation and merging), it is designed for block-access devices like disks as well as for in-memory databases, and it adapts to new query patterns (key ranges of interest) with relatively few queries. Whereas database cracking might process millions of queries before index optimization ceases, adaptive merging finishes B-tree optimization during dozens of queries for the same table and query sequence [7].

Both database cracking and adaptive merging have been described and evaluated to-date primarily in the context of range queries, i.e., predicates of the type “column-value between low-constant and high-constant”. These predicates apply to many non-key columns in relational databases but usually not to key columns. In most cases, a “≤” comparison for keys has no real-world meaning. For example, two people arguing who has the higher social security number would be considered silly. Keys are compared with “=” comparisons including derivatives such as “in” predicates. This difference is often reflected in the data type: while non-key columns are often floating point values, keys are usually integers.

While database cracking applies immediately to key columns, adaptive merging does not. In particular, database cracking can perform range partitioning with a single key supplied by an equality predicate just as well as with two keys supplied by a range predicate. Adaptive merging, however, requires a range with start and end keys for each merge step.

Beyond applying adaptive merging to key columns, we propose two novel performance improvements that both seem much more promising for keys than for non-key columns and range queries. First, partitions within a partitioned B-tree can be created and maintained that take optimal advantage of the memory hierarchy of CPU cache, in-memory buffer pool, flash disk, etc. For example, the smallest partition with the hottest data is kept smaller than the CPU cache, another partition is kept smaller than the buffer pool, another partition is targeted to a flash device, and the remaining data remains on slow, inexpensive, traditional disks. Appropriately guided data movement between partitions ensures that each partition maintains the right size and contents. In other words, adaptive indexing with appropriate improvements enables self-managing and efficient data placement in a memory hierarchy, even one with multiple levels.

Second, adaptive merging can be employed in the context of master-detail clustering using merged indexes [6], a generalization of combined images [10] and join indexes [18]. A successful combination of master-detail clustering and adaptive indexing should ensure that joining and clustering effort focuses on those complex objects and their components actually accessed together. For example, records about customers, orders, invoices, line items, payments etc. may be clustered only for those customers actually queried, e.g., a subset chosen for auditing. Thus, adaptive indexing enables self-managing master-detail clustering. Together, the two new techniques enable caching of join results.

II. Prior Work

We have identified four areas of work related to the proposed designs. In addition to database cracking and adaptive indexing, the following sub-sections discuss index tuning, object caching, and merged indexes. Much of this section is copied and derived from prior work [5, 6]. It might be useful to note that unlike partial indexes [17], an adaptive

[1] Published by 5th International Workshop on Self Managing Database Systems (SMDB 2010), March 1, 2010, Long Beach, California, USA.

index covers all rows but the structure is optimized only when and where needed by actual queries.

A. *Index Tuning*

Index tuning is complementary to adaptive indexing. Both contribute to self-management of database systems. Index tuning might force or prohibit, encourage or discourage specific indexes after analyzing an artificial or actual workload. Adaptive indexing provides mechanisms for index creation and incremental index optimization. It can be employed within the guiding policies determined by index tuning or it may operate independently. In the remainder of this paper, we assume that creation of a specific index is at least desirable.

B. *Database Cracking*

Database cracking combines some features of both automatic index selection and partial indexes. When a column is used in a predicate for the first time, a cracker index is created by copying all data values in the appropriate column from the table's primary data structure. When the column is used in the predicates of further queries, the cracker index is refined until sequential searching a partition is faster than binary searching in the AVL tree guiding a search to the appropriate partition.

The keys in a cracker index are partitioned into disjoint key ranges and unsorted within each. Each range query analyzes the cracker index, scans key ranges that fall entirely within the query range, and uses the two end points of the query range to further partition the appropriate two key ranges. Thus, in most cases, each partitioning step creates two new sub-partitions using logic very similar to partitioning in quicksort [9]. A range is partitioned into 3 sub-partitions if both end points fall into the same key range. This happens in the first partitioning step in a cracker index (because there is only one key range encompassing all key values) but is unlikely thereafter [11]. Updates and their efficient integration into the data structure are covered in [12], and multi-column indexes to support selections and tuple reconstructions are covered in [13].

Recent work [6] has suggested a small improvement to database cracking, based on an improvement of partitioning in quicksort [2]. The core idea is to create a partition with keys equal to the pivot value separate from both lower and higher keys. Thus, a key range is partitioned into 3 sub-partitions using one pivot key or into 5 sub-partitions if two query boundaries fall into the same key range. Doing so enables subsequent equality queries for a previous partitioning key to scan precisely the required records.

A. *Adaptive Merging*

The essence of adaptive merging is to exploit partitioned B-trees [4] in a novel way, namely to focus merge steps on those key ranges that are relevant to actual queries, to leave records in all other key ranges in their initial places, and to integrate the merge logic as a side effect into query execution. Thus, adaptive merging is adaptive and incremental like database cracking. They differ, however, as one relies on merging and the other relies on partitioning, resulting in substantial differences in the speed of adaption to new query patterns.

The differences in query performance are due to data being kept sorted at all times in a B-tree. The difference in reorganization performance, i.e., the number of queries required before a key range is fully optimized, is due to merging with a high fan-in rather than partitioning with a low fan-out of 2 or 3 and to merging a query's entire key range rather than only dividing the two partitions with the query's boundary keys.

When a column is used in a predicate for the first time (and a new index is considered desirable at this time), a run generation algorithm such as quicksort is used to append as many partitions as necessary. Each run forms a partition in the new B-tree. Runs are not merged at this time. Their number depends primarily on input size and memory allocation but also on the sort algorithm and any incidental correlation between the sort order in the data source and in the new index.

The CPU effort for run generation is substantially higher than for predicate evaluation; thus, run generation imposes a substantial penalty in terms of CPU effort on this first query. Judicious memory allocation can control run size, comparison count per record, and thus overall CPU effort. Given today's CPUs, however, the principal cost is in movement in the memory hierarchy, e.g., disk I/O or cache faults. While run generation doubles the movement effort (read-write instead of read-only), it maximizes the benefit for subsequent queries.

When a column is used in a predicate for the second time, an appropriate index exists, albeit not yet fully optimized and merged into a single partition. In this situation, a query must find its required records within each partition, typically by probing within the B-tree for the low end of the query range and then scanning to the high end.

Instead of just scanning the desired key range one partition at a time, however, the query might as well scan multiple partitions in an interleaved way, merge these multiple sorted streams into a single sorted stream, write those records into a new partition within the partitioned B-tree, erase or invalidate the records merged and moved to a new partition, and also return those records as the query result. The data volume touched and moved is precisely that of the query result. A table of contents keeps track of key ranges present or absent in each partition. After a key range has been removed from a partition, this partition will not again be searched for this key range. After very few queries, only a single partition needs to be searched and performance equals that of a traditional B-tree index built separately from and prior to query processing.

B. *Object Caching*

Caching enables fast access to frequently accessed "hot" data items. If caching relies on a buffer pool containing images of disk pages, cold data records may pollute the cache. Thus, the hot items are often copied into a separate memory area together with other hot items. This technique also enables performance benefits with respect to virtual memory, CPU cache, etc. A cache of proper size will, simply by usage, be loaded and retained in the most appropriate level in the memory hierarchy. For example, if a data record is accessed frequently and kept in a data structure smaller than the CPU cache, the standard replacement algorithms in the hardware ensure that this record indeed remains in the CPU cache.

In addition to extracting and concentrating hot data items, data items that are accessed together are often copied together. In relational databases, the concept "together" is usually expressed by foreign key constraints or by repeated usage of equivalent join predicates. Thus, it is desirable to exploit foreign key relationships of hot data items by caching all records related to them. For example, for the most frequent customer of a business, it seems like a good idea to cache that customer's orders, invoices, their line items, etc.

C. *Merged Indexes*

Merged indexes are B-trees that contain multiple traditional indexes and interleave their records based on a common sort order. For example, a single B-tree may contain indexes on customer identifiers in a "customer" table and in an "orders" table, with equal values for customer identifiers enabling co-location of related records.

In relational databases, merged indexes implement "master-detail clustering" of related records, e.g., orders and order details, such that all related records can be retrieved from a single location, e.g., a single disk block. Thus, merged indexes shift de-normalization from the logical level of tables and rows to the physical level of indexes and records, which is a more appropriate place for it. For object-oriented applications, clustering can reduce the I/O cost for joining rows in related tables to a fraction compared to traditional indexes, with additional beneficial effects on buffer pool requirements.

A strict separation of B-tree and index into two layers of abstraction enables a design for merged indexes in which the set of tables, views, and indexes can evolve without restriction. The set of clustering columns can also evolve freely. A relational database can search and update index records just as in traditional indexes. Thus, merged indexes may finally bring general master-detail clustering to traditional databases together with its advantages in performance and cost.

If records from multiple indexes are interleaved within a single B-tree, each record must identify the index to which it belongs. For maximal flexibility, all leading key fields up to and including the index identifier are tagged with their domain. Tags can be represented by very small integers in most cases. For example, in a B-tree that merges orders and their line items based on order number, the sort key consists of 4 to 5 pieces: the tag for the domain "order number," a value from that domain, a tag for the domain "index identifier," a value from that domain, i.e., the actual index identifier, and the line number within an order. The last piece occurs only in line item records. It needs no tag as it follows the index identifier.

If the index identifier is the leading sort key, the record type is separated from all others. In this case, multiple indexes may co-exist in the same B-tree, but their records are not interleaved or clustered based on key values. Prefix truncation [3] avoids practically all storage overhead but this case is rarely more useful than a separate B-tree for each index.

III. Database cracking for keys

For indexes on relational keys, equality predicates can supply boundary keys for range partitioning just as well as range predicates in previous work. Two differences between equality queries and range queries deserve mention.

The first difference is that there is only one new boundary key and new partition for each equality query (not two as for range queries). Thus, it takes twice as many queries before an initially unsorted array is partitioned into small partitions and until an index reaches its final state [11,13].

The second difference is that there is no obvious choice whether the partitioning key should go to the small or the large partition. Lacking an obvious choice, it's best to assign a separate partition to entries equal to the new partitioning key. This design choice also ensures that future equality queries with the same key achieve optimal performance with zero overhead. Thus, an optional improvement for non-key columns and range predicates is rather a requirement for key columns and equality predicates.

IV. Adaptive merging for keys

In prior work, which only considered range queries, adaptive merging was found to finalize index optimization with fewer queries than database cracking. This is partially due to each query optimizing future searches in the query's entire key range, whereas database cracking optimizes only the partitions containing the query's end points. In the case of equality queries, this advantage of adaptive merging must vanish, because the key range is only a single value.

Even in prior work, however, adaptive merging was designed for block-access devices. For a query with a small key range that may extract less than an entire data block from each initial run, it seemed advantageous to merge more than what is strictly required. In fact, prior experiments expanded the key ranges based on both the width of the query's range predicate and the expected record count in a data block.

A similar technique is required when applying adaptive merging to indexes on relational keys and equality predicates. In the experiments reported below, a key range for each data block is estimated from the record count of the workspace, the merge fan-in, and the number of initial runs after run generation. This key range is rounded to maximize suffix truncation as described for prefix B-trees [3]. If an equality query needs to search multiple runs or B-tree partitions, it performs a merge step for the key range thus calculated.

V. Adaptive indexes in a memory hierarchy

Prior work on adaptive indexing as well as the sections above have implicitly assumed a very simple memory hierarchy. Database cracking was invented primarily for in-memory databases, adaptive merging for block-access devices. If a database is stored in a memory hierarchy with multiple levels, additional challenges and opportunities require attention. If performance of self-managing indexes is crucial for their success, exploiting memory hierarchies is critical.

For adaptive merging using partitioned B-trees, one opportunity is to match partitions to levels in the memory hierarchy. For example, if one partition contains only "hot" items that are accessed frequently and is smaller than a given level in the memory hierarchy, e.g., the buffer pool, it is virtually assured that those hot items and their B-tree partition will remain in that level, e.g., the buffer pool.

This leads to adaptive merging with three instead of two stages: initial runs, a single "big" partition, and a single "hot" partition. The new technique "un-merges" data items from one partition, and moves them to another. The implementation of the "move" operation is very simple in a partitioned B-tree; it merely requires updating the partition identifier or artificial leading key field in affected B-tree records.

The number of special partitions of this kind equals the number of levels in the memory hierarchy above mass storage, e.g., the disk. An insertion is applied only in the highest level of the memory hierarchy or, more specifically, the appropriate partition; a successful search moves the found item to the highest level in the memory hierarchy and the appropriate partition; an unsuccessful search inserts a ghost record into that partition; and a deletion inserts an anti-matter record.

Items are moved up when accessed and moved down using a replacement policy such as LRU. When, upon an insertion into a level in the memory hierarchy, the corresponding partition exceeds an appropriate size, some items must be moved down in the memory hierarchy by updating the artificial leading key field. Obviously, a buffer pool is required in addition to the partitions, but the buffer pool at each level can be kept small, just large enough to hide access latencies. For partitions primarily designed to match levels in the memory hierarchy, size limitations should be enforced sufficiently aggressively to avoid "double page faults" [15].

In database indexes on non-key columns, neighboring key values are often accessed together due to range predicates. Overall performance depends more on bandwidth than on latency. For key columns, access latency to specific key values is the crucial performance characteristic. Raising individual key values in the memory hierarchy is the most promising technique for self-managing indexes.

We may also employ a new optimization for adaptive indexing, and insert a "tomb stone" or "ghost record" into the final partition of the B-tree index. Non-existent keys could thus participate in "hot" partitions optimized for the memory hierarchy. Ghost records are a well known and often used technique for key deletion in B-tree indexes [8].

VI. MERGED INDEXES AND ADAPTIVE INDEXES

Merged indexes [5] implement master-detail clustering of related records or entire complex objects based on equal join keys. Merged indexes have not been combined with partitioned B-trees or with adaptive indexing in prior work. This combination opens new opportunities for self-managing indexes and for very efficient database retrieval.

If a merged index is stored in a partitioned B-tree, the partition identifier precedes the sort key within the merged index. Thus, if a merged index is loaded from an unsorted data stream interleaving multiple record types, the in-memory sort algorithm compares records based on key values as used in the merged index and then adds a run number as it emits runs.

If indexes (and thus record types) are loaded one at-a-time, they fit into a merged index using an index identifier as leading index key, immediately following the partition identifier. The additional index key is mostly logical for both partition identifier and index identifier, because prefix truncation [3] avoids practically all storage overhead.

Adaptive merging applied to a merged index permits focusing all reorganization and optimization effort on only those complex objects retrieved by actual queries. If a complex object is accessed for the first time, its components are gathered from the partitions created during the initial load operation. The appropriate records are modified with both a new partition identifier and, if required, the index identifier is moved to trail the user-defined sort key, e.g., order number. Any subsequent access to that object can retrieve all its components from a single location, e.g., a single disk block.

The new techniques described above apply to merged indexes just as well as to "pure" indexes: empty query results can be marked by "tomb stone" records, updates can be appended into new partitions including deletions in the form of "anti-matter" records, and a merged index can be optimized for a memory hierarchy. Thus, the combination of old and new techniques extends adaptive indexing from a mechanism for self-managing traditional indexes to one for self-managing complex objects in a deep memory hierarchy. What started as a research effort to extend adaptive indexing from non-key columns to keys is turning into a technology for self-managing high-performance database systems.

Interestingly, this extension is possible with fairly limited modifications to traditional indexing code: multi-column B-trees enable partitioned B-trees, tagged key values enable master-detail clustering, prefix truncation reduces storage overhead, quicksort and replacement selection enable run generation, and B-trees enable selective retrieval within runs.

VII. PERFORMANCE EVALUATION

Due to lack of space, the performance evaluation here is very short and, admittedly, preliminary.

For a short comparison of database cracking and adaptive merging on keys, assume 10^7 records with distinct values. Leaving these records entirely unsorted requires 10^7 comparisons in each query execution. A traditional B-tree index (created and fully optimized prior to query execution) requires 24 comparisons for each search. In adaptive merging, run generation by replacement selection with an in-memory workspace of 10^5 records produces about 51 runs; 51 searches require $51\times24=1{,}224$ comparisons. This is immediately after run generation; subsequent merging reduces the number of comparisons to 24 when only a single partition requires searching. Database cracking with unsorted range partitions requires at least $10^7\div1{,}224=8{,}169$ partitions or 8,168 equality queries before it can process a new equality query with 1,224 comparisons (on average), i.e., before it can match the initial performance of adaptive merging using partitioned B-trees.

In the following experiments, 10^7 records with distinct key values are queried 2,500 times repeatedly with random search keys across the entire domain. The workspace during run generation fits 10^5 records, the merge fan-in is 100. In terms of balancing the effort for index creation between run generation and merging, each record participates in 17-18 comparisons during run generation and in about 6-7 comparisons when merged, and each record is copied and moved once during run generation and once during merging.

The following diagrams illustrate the overhead of initializing, searching and optimizing adaptive indexes beyond the effort required to search a traditional B-tree index created a priori. Note that our cost metric is focused on movements in the memory hierarchy and it measures the number of records touched; it does not reflect the number of comparisons. Based on the assumption that a B-tree index fully optimized prior to query processing is efficient, the following diagrams show the additional query processing effort due to adaptive indexing. Zero additional effort is the most desirable outcome in these experiments. Not having an index at all, probably in many situations the most realistic alternative to adaptive indexing, imposes an overhead of scanning 10^7 records in each query instead of a single record. Both techniques essentially implement sort algorithms O (N log N) comparisons.

The curves reflect original database cracking (top curve), database cracking with our optimizations, in particular sorting minimal partitions (dashed curve), and adaptive indexing (bottom curve). Each data point averages 1% of the workload, i.e., of 25 queries, including the first data point.

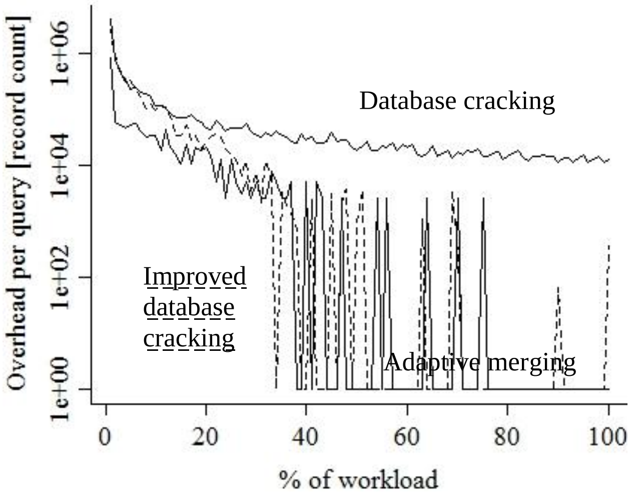

Figure 1. 10^7 distinct key values.

Figure 1 shows how original database cracking burdens many queries with partitioning steps. Even after 2,500 queries (100% of the workload), partitioning continues. Improved database cracking has similar overhead early in the query sequence but once a sub-partition is smaller than the workspace (10^5 records), the entire sub-partition is sorted and future searches can use binary search rather than sequential search. "Spikes" indicate queries in key ranges not yet sorted. Adaptive indexing converges most efficiently towards the performance of a traditional index created a priori. It benefits from the sort effort during the initial copy step (run generation) and from merging entire data blocks around the specific key sought by a query. After about 2,000 queries (80% of the workload), query processing with a partitioned B-tree optimized as a side effect of query execution is as efficient as with a traditional B-tree index created a priori.

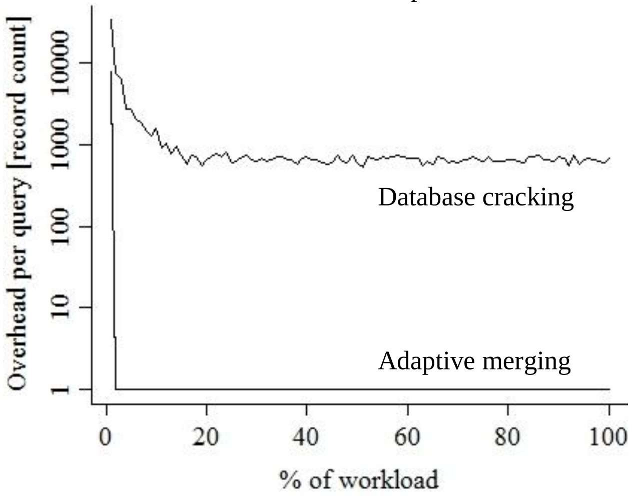

Figure 2. Master index with distinct keys.

Figure 2 shows adaptive indexing for primary keys, i.e., an index of moderate size (10^5 records) with unique keys. As expected, both optimized database cracking and adaptive merging have no overhead compared to a traditional index except in the very first query when the index is created. In fact, their curves are indistinguishable in Figure 2. Adaptive merging generates runs (including sorting in the in-memory workspace) as part of the copy step; optimized database cracking sorts as side effect of the first query. Original database cracking, on the other hand, only performs binary partitioning during each query execution and thus requires a long query sequence before the index is fully optimized.

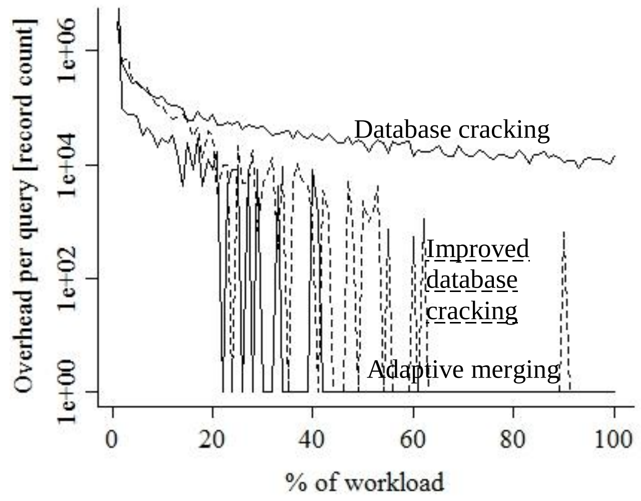

Figure 3. Detail index with 99 records per distinct key.

Figure 3 illustrates the performance of adaptive indexing for foreign keys, i.e., an index on the matching details table. There are 99 detail records for each master record in Figure 2.

Overall, progress towards a fully optimized index and thus overhead per query are similar to the experiment in Figure 1, indicating that the techniques are robust for moderate number of duplicate key values in an index.

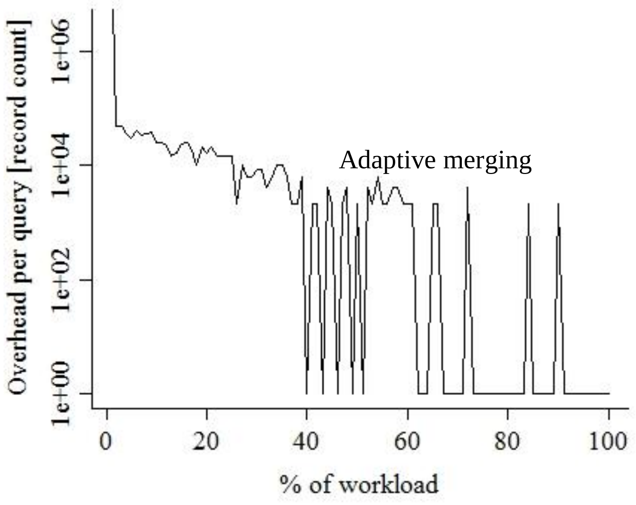

Figure 4. Master-detail clustering − 100 records per key.

Finally, Figure 4 shows the performance of adaptive merging in a merged index with master and detail records, or 100 records per distinct key value. Again, not surprisingly, the swift optimization and reducing of overhead are similar to the experiment in Figure 3.

Due to limits in paper size and the speed of our coding, we cannot report on measurements of benefits and overheads of adaptive indexing optimized for a memory hierarchy.

VIII. Summary and conclusions

In summary, adaptive indexing can be a pillar of self-managing database systems. Both database cracking and adaptive merging apply to non-key columns and range predicates as well as key columns and equality predicates. In online analytical processing, keys correspond to dimensions and non-key columns to measures. Modifying database cracking is trivial. One rather than two new partition boundaries per query imply twice as many queries before index optimization is complete, with completion time already identified as a weakness of database cracking compared to adaptive merging. Adaptive merging for key columns actually requires rounding boundary keys whereas it merely benefits from rounding in the case non-key columns. Adaptive merging for key columns enables self-tuning and self-managing master-detail clustering as well as self-tuning and self-managing optimization of B-trees for memory hierarchies. The performance evaluation demonstrates that all three techniques support equality queries similarly well to range queries, that the proposed optimizations for database cracking reduce the overhead during query execution, and that adaptive indexing converges towards a fully optimized B-tree and adapts to a new query pattern much faster than either variant of database cracking.

In conclusion, index creation and optimization can be entirely automatic for both key columns and non-key columns, and it focuses on key values retrieved by actual queries, avoiding wasted effort on other key values. If desirable, adaptive indexing can be subject to policy decisions by tuning tools that may force or prohibit, encourage or discourage indexes; adaptive indexing provides enabling mechanisms for index creation and optimization as side effect of query execution, i.e., in a very non-intrusive way. Among the two principal techniques for adaptive indexing, adaptive merging implemented using partitioned B-trees exploits available memory and processing power during initial index creation and during subsequent index optimization; thus, index optimization terminates more quickly and optimal query performance is achieved with few queries in any key range. Self-managing optimizations for memory hierarchies and for complex object storage, introduced in the present paper, round out the capabilities of adaptive indexing and of adaptive merging.

Acknowledgment

Stratos Idreos and Stefan Manegold very graciously and generously contributed insightful comments and suggestions to our papers on adaptive merging. Barb Peters suggested multiple improvements in the text.

References

[1] José A. Blakeley, Per-Åke Larson, Frank Tompa: Efficiently Updating Materialized Views. SIGMOD 1986: 61-71.
[2] Jon Louis Bentley, M. Douglas McIlroy: Engineering a sort function. Softw., Pract. Exper. 23(11): 1249-1265 (1993).
[3] Rudolf Bayer, Karl Unterauer: Prefix B-trees. ACM TODS 2(1): 11-26 (1977).
[4] Goetz Graefe: Sorting and indexing with partitioned B-trees. CIDR 2003.
[5] Goetz Graefe: Master-detail clustering using merged indexes. Inform., Forsch. Entwickl. 21(3-4): 127-145 (2007).
[6] Goetz Graefe, Harumi Kuno: Self-selecting, self-tuning, incrementally optimized indexes. To appear in EDBT 2010.
[7] Goetz Graefe, Harumi Kuno: Two adaptive indexing techniques: improvements and performance evaluation. Submitted.
[8] Jim Gray, Andreas Reuter: Transaction processing: concepts and techniques. Morgan Kaufmann 1993.
[9] C. A. R. Hoare: Algorithm 64: Quicksort. Comm. ACM 4(7): 321 (1961).
[10] Theo Härder: Implementing a generalized access path structure for a relational database system. ACM TODS 3(3): 285-298 (1978).
[11] Stratos Idreos, Martin L. Kersten, Stefan Manegold: Database cracking. CIDR 2007: 68-78.
[12] Stratos Idreos, Martin L. Kersten, Stefan Manegold: Updating a cracked database. SIGMOD 2007: 413-424.
[13] Stratos Idreos, Martin Kersten, Stefan Manegold. Self-organizing tuple reconstruction in column stores. SIGMOD 2009: 297-308.
[14] C. Mohan, Inderpal Narang: Algorithms for creating indexes for very large tables without quiescing updates. SIGMOD 1992: 361-370.
[15] Michael Stonebraker: Operating system support for database management. Comm. ACM 24(7): 412-418 (1981).
[16] Dennis G. Severance, Guy M. Lohman: Differential files: their application to the maintenance of large databases. ACM TODS 1(3): 256-267 (1976).
[17] Praveen Seshadri, Arun N. Swami: Generalized partial indexes. ICDE 1995: 420-427.
[18] Patrick Valduriez: Join indices. ACM TODS 12(2): 218-246 (1987).